Pregled i analiza algoritama za sortiranje konačnog niza realnih brojeva

Vručinić, Vlaho

VELEUČILIŠTE
s pravom javnosti
BALTAZAR ZAPREŠIĆ
Zaprešić

Preddiplomski stručni studij
Informacijske tehnologije

VLAHO VRUČINIĆ

PREGLED I ANALIZA ALGORITAMA ZA SORTIRANJE KONAČNOG NIZA REALNIH BROJEVA

STRUČNI ZAVRŠNI RAD

Zaprešić, 2021. godine

VELEUČILIŠTE
s pravom javnosti
BALTAZAR ZAPREŠIĆ
Zaprešić

Preddiplomski stručni studij
Informacijske tehnologije

STRUČNI ZAVRŠNI RAD

PREGLED I ANALIZA ALGORITAMA ZA SORTIRANJE KONAČNOG NIZA REALNIH BROJEVA

Mentor:
mr. sc. Josip Lopatič

Student:
Vlaho Vručinić

Naziv kolegija:
Algoritmi i strukture podataka

JMBAG studenta:
0234055664

SADRŽAJ

SAŽETAK

Algoritmi, matematički načini rješavanja informatičkih problema pojavljuju se u antičko vrijeme, a u 9. stoljeću dobivaju naziv po prezimenu perzijskog matematičara al-Khwārizmī koje latinizirano zvuči *Algoritmija.* Primjena algoritama počela je u matematici da bi se u današnje vrijeme najviše koristila u informatici ali i u disciplini kreiranja i slanja šifriranih poruka, kriptografiji.

Moj rad bavi se teorijskom prirodom algoritama a pitanju funkcije algoritama i načina njihovog funkcioniranja pristupam komparativno i analitički. Pojašnjene su funkcije algoritama, a kroz praksu su na primjerima prikazani načini funkcioniranja algoritama u određenim programskim jezicima. Usporedbom algoritama po parametrima poput efikasnosti, brzine te vremenske kompleksnosti povezat ću teoriju algoritama s njihovim značenjem za prosječnog čovjeka u svakodnevnom životu. Prikazom šest algoritama pokazat ću njihove osnovne značajke. Primjerice pokazat ću kako je *Heap sort*, prvi algoritam u ovom radu, vrlo brz i efikasan a temelj mu je binarno stablo. *Quick sort* je također brz i efikasan, jako koristan, a metoda njegovog rada je „divide and conquer". *Insertion sort* je koristan algoritam u određenim situacijama, kod malih nizova, stabilan je, međutim nije baš efikasan. *Merge sort* je jako brz, koristi „divide and conquer metodu" jednako kao *quick sort*, međutim iako jednostavan i brz, u manjim nizovima je spor. *Shell sort* „in place" metoda, je varijacija na *Insertion sort*, ali je jednostavniji i optimiziran. *Bubble sort* jako jednostavan, ograničene je uporabe samo na edukacije o algoritmima, te spor je i dugačak.

Ovaj rad donosi usporedbu navedenih šest algoritama iz koje je jasno vidljivo da svaki od algoritama najbolje funkcionira u određenim situacijama te ni za jednog ne možemo reći da je najbolji ili najlošiji nego su podjednako dobri ili manje dobri ovisno o zahtjevima situacije u kojima se koriste.

Ključne riječi: niz, polje, red, vremenska kompleksnost, sortiranje

Title in English: REVIEW AND ANALYSIS OF ALGORITHMS FOR SORTING A FINITE SEQUENCE OF REAL NUMBERS

ABSTRACT

Algorithms, mathematical ways of solving computer problems appear in ancient times and in the 9th century were named after the surname of the Persian mathematician al-Khwārizmī, which in Latin sounds like Algorithm. The application of algorithms began in mathematics, and nowadays it is mostly used in informatics, but also in the discipline of creating and sending encrypted messages, cryptography.

My paper deals with the theoretical nature of algorithms and I approach the question of the function of algorithms and the way they work in comparative and analytical manner. The functions of algorithms are explained, and through practice, the ways of functioning of algorithms in certain programming languages are shown thru examples. By comparing algorithms by parameters such as efficiency, speed and time complexity, I will connect the theory of algorithms with their significance for the average person in everyday life. By showing six algorithms I will show their basic features.

For example, I will show how the *Heap sort*, the first algorithm in this paper, is very fast and efficient and is based on a binary tree. *Quick sort* is also fast and efficient, very useful, and the method of his work is "divide and conquer". *Insertion sort* is a useful algorithm in certain situations, in small arrays. It is stable, but it is not very efficient. *Merge sort* is very fast, it uses the "divide and conquer method" as well as the *quick sort*, but although simple and fast, it is slow in smaller series. The *shell sort* "in place" method is a variation on the *Insertion sort*, but is simpler and optimized. *Bubble sort* is very simple, its limitation is in its predominant use in education about algorithms. It is slow and long.

This paper brings a comparison of these six algorithms from which it is clear that each of the algorithms works best in certain situations and none can be said to be the best or worst but are equally good or less good depending on the requirements of the situation in which they are used.

Key words: sequence, array, queue, time complexity, sort

1. UVOD

Algoritam kao pojam i funkcija postoji još od antičkog razdoblja pa ga tako nalazimo, primjerice, u Grčkoj kod atenskog matematičara Euklida koji je osmislio algoritam za pronalaženje najvećeg zajedničkog djelitelja dva broja. U poznatije jednostavne algoritme uvrštava se i Eratostenovo sito koje se koristi za pronalaženje svih prostih brojeva koji postoje do određenog, proizvoljno odabranog broja. Sam naziv algoritam potječe od prezimena perzijskog matematičara iz 9. stoljeća: Muhammada ibn Mūse al-Khwārizmīja. Naime, latinizirana inačica njegova prezimena (al-Khwārizmī), koja označava njegovo porijeklo, glasi „Algoritmija". Prema autoru jednog članka vezanog uz povijest i razvoj algoritama: „Algorithms as we know today were only put into place with the advent and rise of mechanical engineering and processes. In its original form, algorithms gave base to the algebra of logic, using variables in calculations."[1](SouvikDas,2016)

Algoritmi su svoju svrhu u velikoj mjeri pronašli i u kriptografiji, znanstvenoj disciplini koja se bavi slanjem informacija pomoću šifriranih poruka. Tako se arapski filozof Al-Kindi uz svoje razne druge suvremenike koristio algoritmima za kriptoanalizu, odnosno za dešifriranje kodiranog sadržaja. U poznatije primjere primjene algoritama u kriptoanalizi iz bliže povijesti ubrajaju se Turingovi strojevi, točnije stroj nazvan „Bomba" koji je osmislio britanski matematičar Alan Turing sa svojim kolegama za vrijeme Drugog svjetskog rata kako bi „razbio" kodove Enigme, skupine strojeva koje je koristila njemačka vojska, pretežno za kodiranje, ali i dekodiranje sadržaja.

Iako su se u prošlosti algoritmi koristili samo u matematici, danas se najčešće susreću i primjenjuju u informatici. Definiraju se kao skupovi određenih naredbi za izvršavanje nekog programa. Ukratko, algoritmi se danas koriste za računalne operacije, obradu podataka, automatizirano zaključivanje te za razne druge zadatke. Oni su, najjednostavnije rečeno, matematički načini rješavanja informatičkih problema.

Izabrao sam ovu temu za svoj rad jer smatram da algoritmi čine temelje informatike te su općenito česta pojava u svakodnevnom životu, iako ljudi toga najčešće nisu svjesni. Moj je rad teorijske prirode te ću problemu funkcije algoritama i načinu njihovog funckioniranja pristupiti komparativno i analitički. U radu ću objasniti kako funkcioniraju te zašto se koriste algoritmi. Kroz teoriju ću prikazati obrazloženje funkcije algoritama, a kroz praksu ću prikazati kako oni funkcioniraju kada se iskoriste u nekom programskom jeziku.

Nakon analize algoritama koje ću predstaviti u svome radu, usporedit ću prikazane algoritme po parametrima poput efikasnosti, brzine te vremenske kompleksnosti. Smatram da je važno obraditi ovu temu za završni rad jer je za svakodnevni život čovjeka u modernom, digitalno usmjerenom svijetu prosječnom čovjeku važno je razumjeti da su sve usluge kojima se koristimo na interentu i kroz razne aplikacije zapravo temeljene na algoritmima. Naročito je

[1] „Algoritmi kakve danas znamo osmišljeni su tek s pojavom i usponom strojarstva i procesa vezanih uz strojarstvo. U svom izvornom obliku, algoritmi su bili osnova algebre logike koristeći varijable u izračunima." (Prijevod autora)

ova tema pregleda i analize algoritama važna za mlade osobe koje se prepoznaju ili upoznaju STEM područje u srednjoj ili višim razredima osnovne škole.

U računarstvu je algoritam sredstvo za rad s podacima, koje funkcionira logikom zadanom od proizvođača. Programiranje je način korištenja algoritama za obradu podataka i rješavanje problema. Temelji se na setu uputa kojima se algoritam vodi u zadanom programskom jeziku. Iako algoritmi ne ovise o jeziku i uvijek su isti, ipak ga različiti programski jezici u svom jeziku interpretiraju.

Da bi algoritam bio dobar i koristan, mora biti jednostavan, imati precizan cilj koji se pokušava provesti, mora imati konačan broj naredbi koje se izvršavaju kako bi kod izvršenja zadnje naredbe stao. Svaki algoritam u sebi mora imati input i output, dakle početak gdje se unose podaci ili naredbe, i kraj gdje se izvršavaju. Tipični oblik algoritma izgleda otprilike ovako:

1. Definicija problema,
2. Razvoj modela,
3. Specifikacija algoritma,
4. Izrada algoritma,
5. Provjera ispravnosti algoritma,
6. Analiza algoritma,
7. Provedba algoritma,
8. Testiranje programa,
9. Priprema dokumentacije,
10. Izvođenje.

Ovaj rad bavit će se sort algoritmom. To je algoritam koji se koristi za sortiranje podataka, koji su obično ili brojevi ili slova, odnosno riječi u nekom redoslijedu. Redoslijed može biti alfabetski ili, primjerice, po veličini brojeva. Sortiranje je jako važno za efikasnost programa odnosno drugih algoritama jer dobro sortiranje može značajno ubrzati druge procese. Postoje razni sort algoritmi. Neki od algoritama kojima se bavi ovaj rad su *bubble, merge, shell, heap, quick i insertion sortovi*. Specifičnije, baviti ću se sortiranjem konačnog niza realnih brojeva. Pregledom, analizom i usporedbom navedenih sortova, kako funkcioniraju i koliko su efikasni, usporedit ću ih u područjima primjene algoritama kako bih vidio njihove prednosti i nedostatke.Namjera mi je ovom usporedbom pružiti svrhovitu informaciju o svakom od njih što bi, po mom mišljenju moglo biti od koristi prosječnom građaninu ali i stručnjacima koji ih koriste u svom radu kako bi prepoznali za koji određeni posao koristiti koji sort.

2. ALGORITMI

2.1. HEAP SORT

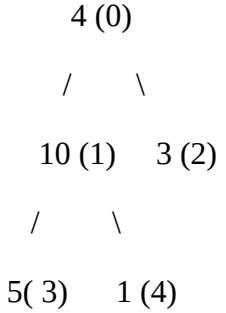
Heap sort je vrsta sortiranja koja funkcionira na način da sortira hrpu kroz binarno stablo. Djeluje tako što podijeli podatke na sortiranu i nesortiranu stranu i najveći broj sa nesortirane strane premješta na sortiranu stranu što se ponavlja sve dok niz nije posložen.

Binarno se stablo sastoji od čvorova gdje je prvi čvor roditeljski, a svi ispod njega su njemu podležni, odnosno manji, čvorovi, kao djeca kod obiteljskog stabla. U binarnom je stablu vrijednost koja se nalazi u roditeljskom čvoru uvijek veća (ako se radi o max sortu), odnosno manja (ako se radi o min sortu) u odnosu na čvorove koji se nalaze ispod njega (podležni čvorovi - djeca). Kod min sorta redoslijed je uzlazan. Dakle, vrijednost broja u prvom, najnižem čvoru je najmanja, a vrijednost svakog idućeg broja je veća u odnosu na prethodni. Max sort je obrnut te ide silaznim redoslijedom tako da počinje najvećim brojem u najnižem čvoru, a svaki idući je manji. Funkcija *heapify*, koja se koristi u ovom primjeru, radi tako što prođe kroz cijelo stablo i roditeljske čvorove te ih uspoređuje s čvorovima koji se nalaze ispod prethodno spomenutog čvora i mijenja ih po potrebi, odnosno ako su čvorovi koji se nalaze ispod čvora s kojim se uspoređuje manji, onda oni ostaju na svom mjestu i ne mijenjaju se, a ako su veći onda se mijenjaju s roditeljskim čvorom ovisno o tome je li redoslijed uzlazan ili silazan (opisano je za silazni niz).

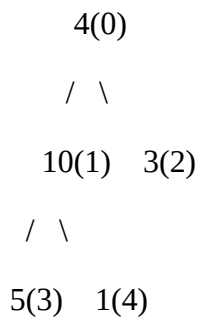
Primjer *heap sorta*:

Input data: 4, 10, 3, 5, 1

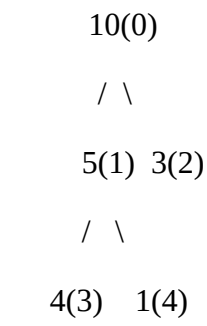
1.korak – Kreira se potpuno binarno stablo na način da se svi brojevi dani u inputu postave u čvorove, po nivoima, redom, slijeva nadesno.



2.korak – Na drugi redni broj 10, čija je vrijednost u čvoru (1), primjenjuje se *heapify* funkcija, odnosno, gleda se je je li vrijednost u čvoru manja ili niža u odnosu na niže čvorove. Kako je broj 10 veći i od broja 5 i od 1, radi se o max sortu, odnosno redoslijed je silazan te 10, kao i 5 i 1 ostaju na svome mjestu.



3.Korak - *Heapify* funkcija se primjenjuje na prvi redni broj, tj. na broj 4; odnosno, gleda se je li manji od vrijednosti čvorova na nivou ispod. Kako je 4 manje od 10, te je ujedno 10 veći 3, a radi se o max sortu, odnosno silaznom, tako 10 mijenja 4 kao glavni čvor. Nadalje, *heapify* funkcija se primjenjuje na čvor (1) u kojem je vrijednost 4, te na isti način 4 i 5 mijenjaju mjesta i sve je sortirano.



Heap sort je brz i efikasan algoritam. Vrlo je jednostavan što je ujedno i razlog njegovoj čestoj upotrebi. Posebno se pokazuje dobro upotrebljiv za velike nizove. U odnosu na druge algoritme, nijedan nije toliko brz i toliko dobro izvodljiv poput *heap sorta*.

S druge strane, problem kod *heap sorta* je što nije „stabilan“, odnosno operacije tijekom sortiranja mogu poremetiti stablo i redoslijed kada su u pitanju isti elementi u provođenju sortiranja, što ga može usporiti. U tom slučaju drugi algoritmi mogu biti brže izvedeni.

2.2. Implementacija HEAP SORTa u C++

Implementacija *heap sorta*:

```
void heapify(int arr[], int n, int i)
{
   int largest = i; // roditeljski element je najveći
   int l = 2 * i + 1; ljevi dio niza = 2*i + 1
   int r = 2 * i + 2; // desni dio niza = 2*i + 2

   // ako je lijevi dio niza veći od roditelja
   if (l < n && arr[l] > arr[largest])
      largest = l;

   // Ako je desni potomak veći od drugih
   if (r < n && arr[r] > arr[largest])
      largest = r;

   // Ako najveći element nije roditelj
   if (largest != i) {
      swap(arr[i], arr[largest]);

      // Heapify funkcija se izvršava na tom određenom podstablu
      heapify(arr, n, largest);
   }
}
// Glavna funkcija za izvršavanje heap sorta
void heapSort(int arr[], int n)
{
```

```
    // Gradi se hrpa i niz se sortira, izvodi se heapify
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Elementi se izdvajaju iz hrpe
    for (int i = n - 1; i > 0; i--) {
        // Trenutni roditelj se miče na vrh niza
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

2.3. QUICK SORT

Quick sort je metoda sortiranja koju je razvio britanski znanstvenik Tony Hoare 1959. godine. Jedan je od najpoznatijih algoritama za sortiranje, a često se koristi zato što može biti brži od *heap sorta* i nekih drugih načina sortiranja, ako se kôd dobro postavi. Funkcionira tako da se izabere jedan broj koji je pivot. Pivot se uspoređuje s drugim brojevima koji se dijele u dvije podgrupe ovisno o tome jesu li veći ili manji u odnosu na pivot. To se uspoređivanje ponavlja sve dok svi elementi nisu na mjestu ovisno o veličini. Odnosno, brojevi koji su manji ili jednaki od pivota smještaju se na lijevu stranu, a oni koji su veći od pivota na desnu stranu, isto kao kod *heap sorta*. Na taj se način zapravo kreiraju dvije podgrupe kod kojih se ponavlja isti postupak – odabire se pivot, uspoređuje se s elementima u nizu te nastaju nove podgrupe. Svi se ti koraci ponavljaju sve dok niz nije potpuno posložen. Pivot može biti bilo koji element, ali obično se uzima prvi ili zadnji. *Quick sort* koristi strategiju „podijeli pa vladaj“ (engl. divide-and-conquer), odnosno dijeli početni, složeniji (veći) niz na sve jednostavnije (manje) koje je lakše riješiti. Na kraju zapravo spojimo liste koje smo konačno dobili dijeljenjem i uspoređivanjem kako bismo kreirali sortiranu listu. *Quick sort* je moguće obavljati i na način da je izabrani pivot srednji član liste. Tada se on uspoređuje s članovima s njegove lijeve i desne strane. Ako je član s lijeve strane manji ili jednak od pivota, ide se dalje, a ako je veći zamijeni se s članom s desne strane pivota. Ako je desno od pivota veći broj, opet se samo nastavlja, a ako je manji zamjenjuje ga se članom s lijeve strane. Tako se postupno provjeravaju lijeva i desna strana sve dok vrijedi uvjet da je indeks lijevog člana manji ili jednak indeksu desnog. Tada se rade dvije manje liste od zadane liste i u njima se ponavlja cijeli postupak.

Primjer *Quick sorta*:

Korak 1 - Postave se elementi

3 6 4 8 1 5

Korak 2 - 5 se uzme kao pivot jer je zadnji element niza i krećemo gledati elemente slijeva na desno. Ako je element manji ili jednak od pivota, ostaje na mjestu. Ako nije, onda se veći pivot i taj element zamijene.

3 je manji pa ostaje na mjestu, 6 je veći pa se pivot (5) i 6 zamjenjuju

3 5 4 8 1 6

Sada gledamo s desne strane na lijevu jer smo imali promjenu i ako je element manji ili jednak od pivota ih mijenjamo

6 je veći, dakle ostaje na mjestu, 1 je manji, dakle mjenja se s pivotom

3 1 4 8 5 6

Sada se dalje gleda slijeva, 8 je veći od 5 pa se mjenjaju

3 1 4 5 6 8

Sada kada je pivot na mjestu, gledaju se podgrupe odnosno lijevo od pivota i desno. Ljevi niz:

3 1 4

Uzmemo 4 kao pivot

3 je manji od 4, isto kao 1, dakle pivot je na mjestu i niz se dijeli dalje

3 1

1 se uzme kao pivot

1 je manji od 3, dakle zamijene se

I lijevi niz je 1 3 4

Desni niz:

8 6

6 je pivot

8 je veći od 6, dakle mijenjaju se

Desni niz je 6 8

Lijevi i desni sortirani niz se vrate u glavni niz koji sada izgleda ovako:

1 3 4 5 6 8

Primjer *Quick sorta* kada je pivot srednji član:

Korak 1 – Kreiramo listu.

12 4 35 6 1 17 5

Korak 2 – Odabiremo srednji član.

Srednji član = 6

Korak 3 – Uspoređujemo elemente s pivotom (6). Broj 12 veći je od 6 pa ga mijenjamo s brojem 5, koji je manji u odnosu na 6. Zatim gledamo 4 i 17, njih ne treba zamijeniti. Nakon toga su 35 i 1 koje treba zamijeniti. Dobivamo sljedeću listu:

5 4 1 6 35 17 12

Korak 4 – Listu oko pivota dijelimo na dvije podliste te imamo:

5 4 1 6 35 17 12

Korak 5 – U prvoj listi biramo, primjerice, 4. Broj 6 može ostati, ali 5 je veći od 4 pa kad se s desne strane pomaknemo na broj 1, koji je manji od 4, moramo zamijeniti 5 i 1. Tako dobivamo:

1 4 5 6

Korak 6 – U drugoj listi uzimamo 17 kao pivot. Vidimo da trebamo zamijeniti 35 i 12. Kada to učinimo, dobivamo:

12 17 35

Korak 7 – Spajamo dvije liste te je konačna lista:

1 4 5 6 12 17 35

2.4. Implementacija QUICK SORTa u C++

```
// Funkcija za zamjenu dva elementa

void swap(int *a, int *b)

{

   int t = *a;

   *a = *b;

   *b = t;

}


//Ova funkcija uzima zadnji element kao pivot, i stavlja taj pivot na svoje mjesto u sortiranoj listi, a sve elemente manje od pivota stavlja na lijevo od njega, a elemente veće od pivota stavlja na desnu stranu //

int partition (int arr[], int low, int high)

{

   int pivot = arr[high]; // postavlja se pivot

   int i = (low - 1); // Postavlja se znak za elemente manje od pivota i pokazuje sve elemente desno od pivota:

   for (int j = low; j <= high - 1; j++)

   {

      // Slučaj kada je element koji se gleda manji od pivota:

      if (arr[j] < pivot)

      {

         i++; // znak za povećanje manjeg elementa

         swap(&arr[i], &arr[j]);  // zamjena elemenata

      }

   }

   swap(&arr[i + 1], &arr[high]);

   return (i + 1);
```

```
}

// glavna funkcija koja unosi i izvršava quick sort:
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

//Elementi se zasebno sortiraju prije i poslije podijele
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

2.5. INSERTION SORT

Insertion sort je metoda sortiranja liste koja funkcionira tako što se uspoređuje jedan element za drugim. Zbog toga zapravo nije naročito efikasna niti brza (poput *merge sorta*, *quick sorta*, *heap sorta*) pa nije toliko korisna za sortiranje velikih nizova, ali je vrlo jednostavna i stabilna pa je pogodna za manje nizove. Radi na način da se odabere prvi element koji u tom trenutku predstavlja najveći element, ali kad se nađe element veći od njega, oni zamijene mjesta te se od tog trenutka uspoređuje novi najveći element. Dakle, zapravo djeluje poput traženja najvećeg elementa koji se postepeno pomiče na kraj liste. Prvi se element uspoređuje s drugim (jer lijevo od prvog nema elemenata), ako je prvi veći od drugog, zamijene im se mjesta. Ako je manji ili jednak, ostaje na mjestu te se tada drugi element uspoređuje s onim koji se nalazi na mjestu nakon njega. Tako svaki put kada naiđemo na element manji od elementa s kojim trenutno uspoređujemo, mijenjamo mjesta te zadržavamo veći element i nastavimo uspoređivanje. Kada mijenjamo mjesta, element koji stavljamo ispred moramo usporediti i s elementima koji su bili prije onog elementa s kojim je promijenio mjesta kako bismo mu mogli naći odgovarajuće mjesto u nizu. Postupak traje sve dok niz nije u potpunosti sortiran te je svaki element na svom mjestu.

Primjer *insertion sorta*:

Nesortirani niz

14 33 27 10 35 19 42 44

Insertion uzima prva 2 elementa i uspoređuje ih:

14 33 27 10 35 19 42 44

14 i 33 su dobro sortirani. Kako je 14 manji, on ostaje na mjestu, a 33 se uspoređuje sa sljedećim elementom, dok 14 ostaje u sortiranoj pod listi.

14 33 27 10 35 19 42 44

27 je manji od 33, a veći od 14 dakle 33 i 27 mijenjaju mjesta. Radi se pregled sortirane podliste i kako je 27 veći od 14, oboje ostaju na svojim mjestima.

14 27 33 10 35 19 42 44

Sada se uspoređuju 33 i 10, 33 je veći dakle 10 i 33 se mijenjaju:

14 27 10 33 35 19 42 44

10 i dalje nije na svom mjestu, jer je u podlisti broj 27 koji je veći od njega, dakle mijenjaju se

14 10 27 33 35 19 42 44.

10 i dalje nije na svom mjestu jer je manji od 14 i opet dolazi do promjene u podlisti:

10 14 27 33 35 19 42 44

33 se uspoređuje dalje i manji je od 35 dakle na svom mjestu je. Lijevo od njega je sortirani niz, a desno je nesortirani, tako dagledamo dalje:

10 14 27 33 35 19 42 44

33 je veći od 19 pa se mijenjaju:

10 14 27 33 19 35 42 44

19 je manji od 33 pa se mijenjaju:

10 14 27 19 33 35 42 44

27 je veći od 19 pa se mijenjaju:

10 14 19 27 33 35 42 44

Sada su svi elementi liste na svom mjestu i proces staje.

2.6. Implementacija INSERTION SORTa u C++

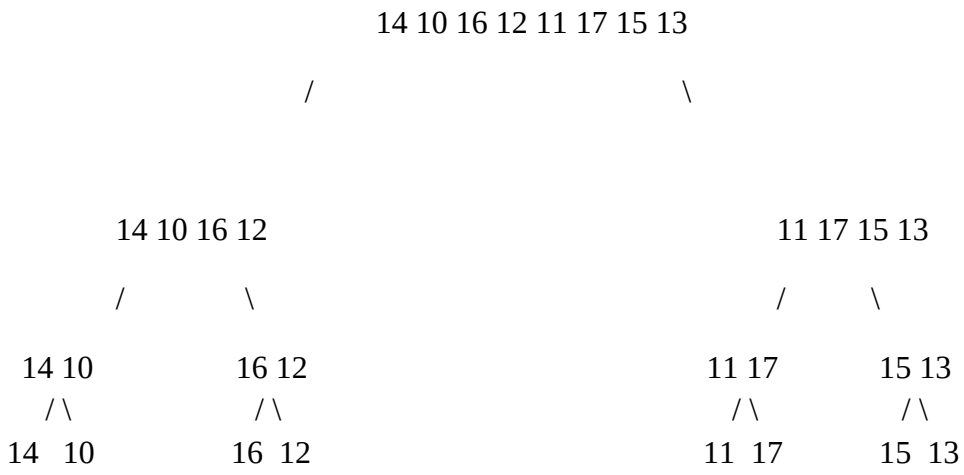
```
//Funkcija za sortiranje niza sa insertion sortom

void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;

      //Miče elementa iz niza arr[0...i-1] od prvog(0) do predzadnjeg(i-1) za jedno mjesto
ako su veći od key elementa:

      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

2.7. MERGE SORT

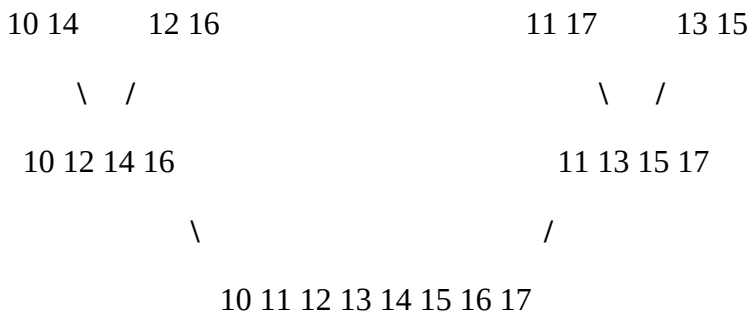
Merge sort se uvrštava među najbrže algoritme za sortiranje. Vrlo je sličan *quick sortu* po tome što također dolazi do dijeljenja početne liste, zasebnog sortiranja te konačnog spajanja *merge*() funkcijom kako bismo dobili sortiranu početnu listu. Odnosno, poput *quick sorta* koristi strategiju „podijeli pa vladaj“. Funkcionira na način da se početna nesortirana lista, koja sadrži „n“ elemenata, podijeli prvo na dvije podliste. Isto se radi sa svakom od novonastalih listi te sa svakom listom koja nastaje ponovnim dijeljenjem. Ovaj se postupak ponavlja sve dok se na kraju ne dobije„n“ zasebnih listi koja se svaka sastoji od jednog člana („n“ elemenata početne liste →„n“ listi s po jednim elementom početne liste). Kako svaka podlista ima samo jedan član, zapravo je svaka sortirana. Nakon što se kreira „n“ jednočlanih podlisti, one se postupno spajaju tako što se međusobno uspoređuju članovi. Time nastaju nove, sve veće i veće liste koje se spajaju *merge*() funkcijom. Kada je cijeli postupak gotov, te se zadnje dvije podliste spoje u jednu listu, dobiva se sortirana verzija početne liste.

1. Podjela niza

Početni nesortirani niz



2. Nesortirani niz je podijeljen na sortirane nizove koji se sastoje od jednog elementa. Sada se ti nizovi spajaju s *merge*() funkcijom.



3. Niz je sortiran.

2.8. Implementacija MERGE SORTa u C++

```
//Spaja dva podniza iz glavnog niza.
// Prvi podniz je arr[begin...mid]
// Drugi podnizje arr[mid+1...end]
void merge(int array[], int const left, int const mid, int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Naprave se privremeni nizovi:
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    // Kopiraju se podaci u privremene nizove leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];
     auto indexOfSubArrayOne = 0, // Prvi element prve podliste
        indexOfSubArrayTwo = 0; // Prvi element druge podliste
    int indexOfMergedArray = left; // Početni indeks spojenog niza



    // Privremeni nizovi se spajaju u array[left..right]
    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo)
{
```

```
        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Elementi koji su ostali u left[] podnizu se prekopiraju ako ih ima:
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Elementi koji su ostali u right[] podnizu se prekopiraju ako ih ima:
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
}

// begin je za lijevi index a end je za
// desni index podniza
// za cijeli sortirani niz
```

```
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Povratak rekurzivno

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

2.9. SHELL SORT

Shell sort je „in place“ metoda sortiranja što znači da ne koristi dodatni prostor za spremanje elemenata koji se sortiraju sa strane. Algoritam je osmislio američki informatičar Donald Shell 1959.godine. Napravljen je kao varijacija insertion sorta, ali je optimiziran tako da radi brže i bolje, iako oba algoritma funkcioniraju na istom principu. Način na koji radi *insertion sort* je da se elementi mijenjaju za jedno mjesto. Dakle, kada se neki element treba pomaknuti za više mjesta, taj je proces kompliciran, zahtjevan i dugotrajan jer ga je potrebno provoditi u više koraka. A ideja *shell sorta* je da to pojednostavi. Metoda po kojoj radi *shell sort* je da se uspoređuju dva (ili više) elementa koji nisu jedan uz drugog, nego su podijeljeni razmakom od n/2 drugih elemenata liste. Pri usporedbi se po potrebi elementima mijenjaju mjesta, ako je sve u redu s položajem članova, pomiče se za jedan element udesno te se tada oni međusobno uspoređuju. Kada se s jedne strane pomakne do kraja, uspoređivanje kreće od početka kako bi se usporedili svi označeni elementi. To se ponavlja u nekoliko prolaza pri čemu se razmak smanjuje po pola. Nakon što se u potpunosti prođe kroz listu, može se reći da je ona sortirana.

Primjer *shell sorta:*

Nesortirani niz

Uzet ćemo n = 9 kao broj elemenata u nizu:

15 19 20 38 24 41 30 31 12

Sada tražimo razmak u elementima koji nije veći od 9:

Razmak = n/2

= 9/2

=4.5

Dakle, za prvi prolaz razmak u elementima je 4.

Uzmemo 0 element, stavimo razmak, dakle gledamo 4. element i još 8. jer dodamo razmak od 4 na 4. element isto kao što na 0. dodamo razmak od 4 i uspoređujemo od prvog označenog elementa pa nadalje:

˅ ˅ ˅

15 19 20 38 24 41 30 31 12

Gledamo ako je 15 veći od 24. Ako je prvi element u usporedbi manji od drugog, kao u ovom slučaju, ne mijenjaju se i samo idemo na sljedeća dva elementa.

Sada gledamo 19 i 41:

1519 20 38 24 41 30 31 12

Opet je ista situacija tako da se ništa ne mjenja.

Idemo dalje:

1519 20 38 2441 30 31 12

20 je manji od 30 tako da gledamo dalje:

1519 20 38 24 41 30 31 12

38 je veći od 31 tako da se zamijene

15 19 20 31 24 41 30 38 12

Vraćamo se na početak i idemo ponovno uspoređivati elemente s udaljenošću od 4.

Označimo elemente s razmakom od 4, ovaj put uspoređujemo od 2. označenog elementa do 3.

15 19 20 31 24 41 30 38 12

24 je veći od 12 tako da se mijenjaju:

15 19 20 31 12 41 30 38 24

Sada uspoređujemo 15 i 12:

15 je veći od 12 tako da se mijenjaju:

12 19 20 31 15 41 30 38 24

Svi označeni elementi su se usporedili i na mjestima su tako da je vrijeme za drugi prolaz.

Za drugi prolaz ćemo uzeti da je razmak 4/2.

Dakle razmak je sada 2:

12 19 20 31 15 41 30 38 24

12 je maji od 20 tako da se ništa ne mijenja i mičemo se za jedno mjesto:

12 19 20 31 15 41 30 38 24

19 je manji od 30 tako da idemo jedno mjesto naprijed:

12 19 20 31 15 41 30 38 24

Prvi element smo već uspoređivali tako da idemo od drugog(20) na dalje.

20 je veći od 15 tako da se mijenjaju:

12 19 15 31 20 41 30 38 24

Sada gledamo lijevu stranu:

12 19 15 31 20 41 30 38 24

12 nije veći od 15 tako da se ništa ne mijenja i mičemo se jedan element dalje:

12 19 15 31 20 41 30 38 24

31 je manji od 41 pa se ništa ne mijenja nego idemo jedno mjesto dalje:

12 19 15 31 20 41 30 38 24

20 nije manji od 30 pa se ništa ne mijenja nego se ide dalje:

12 19 15 31 20 41 30 38 24

41 je veći od 38 pa se mijenjaju:

12 19 15 31 20 38 30 41 24

Sada gledamo lijevi dio niza:

12 19 15 31 20 38 30 41 24

31 je manji od 38 pa se ništa ne mijenja i idemo dalje:

12 19 15 31 20 38 30 41 24

30 je veći od 24 pa se mijenjaju i gledamo red s lijeva:

12 19 15 31 20 38 24 41 30

Sad smo prošli sve elemente i idemo u 3. prolaz

12 19 15 31 20 38 24 41 30

Još smanjimo razmak, trenutni je bio 2 , razmak = n/2.

Razmak = 1

12 19 15 31 20 38 24 41 30

Sada kada je razmak 1 sortiranje funkcionira isto kao *insertion sort*

Sada uspoređujemo slijeva na desno

Prvo gledamo 12 i 19, oni su na točnim mjestima i ne mijenjaju se, gledamo dalje:

12 19 15 31 20 38 24 41 30

Uspoređuju se 19 i 15, 19 je veći pa se mijenjaju:

12 15 19 31 20 38 24 41 30

Provjerimo stranu ljevo od elementa koje smo promijenili:

12 je manji od 15 i na svom mjestu je, možemo uspoređivati dalje:

12 15 19 31 2038 24 41 30

19 je manji od 31 i ništa se ne mijenja, nego idemo jedno mjesto dalje.

12 15 19 31 20 38 24 41 30

31 je veći od 20 pa se mijenjaju.

Sada provjerimo vrijednost ljevo od nove:

12 15 19 20 31 38 24 41 30

Oba elementa su na svojim mjestima i idemo dalje:

12 15 19 20 31 38 24 41 30

30 je manji od 38 i ostaje na svom mjestu.

12 15 19 20 31 38 24 41 30

38 je veći od 24 tako da se mjenjaju i gleda se element lijevo od 24.

12 15 19 20 31 24 38 41 30

31 je veći od 24 i dolazi do promjene i opet se gleda element lijevo od 24.

12 15 19 20 24 3138 41 30

20 je manji i ostaje na svom mjestu, vraćamo se na 38 i gledamo njegovu desnu stranu:

12 15 19 20 24 31 38 41 30

38 je manji od 41, ostaje na svom mjestu i mičemo usporedbu za jedno mjesto.

12 15 19 20 24 3138 41 30

41 je veći od 30 i mijenjaju se, a nakon promjene 30 se uspoređuje s elementom lijevo od njega.

12 15 19 20 24 313830 41

38 je veći od 30 i dolazi do promjene i gleda se sljedeći element slijeva:

12 15 19 20 24 3130 38 41

31 je veći od 30 i mijenjaju se

12 15 19 20 2430 3138 41

24 je manji od 30 i na svom mjestu je.

Svi elementi su na svojem mjestu i niz je sortiran.

2.10. Implementacija SHELL SORT u C++

Shell sort

// postavljanje funkcije za sortiranje niza uz *shellsort*:

```
int shellSort(int arr[], int n)
{
   // Počne se s velikim razmakom, koji se postupno smanjuje:
   for (int gap = n/2; gap > 0; gap /= 2)
   {
      for (int i = gap; i < n; i += 1)
      {
         int temp = arr[i]
         int j;
         for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            arr[j] = arr[j - gap]
         arr[j] = temp;
      }
   }
   return 0;
}
 void printArray(int arr[], int n)
{
   for (int i=0; i<n; i++)
      cout << arr[i] << " ";
}
```

2.11. BUBBLE SORT

Bubble sort je najjednostavniji algoritam za sortiranje podataka te ga je ujedno vrlo lako koristiti. Funkcionira na principu usporedbe dva susjedna elementa. Naime, kroz svaki se prolaz uspoređuju dva po dva elementa te im se mijenjaju mjesta, ako nisu dobro posloženi po veličini. Prolazi se kroz listu sve dok ona nije u potpunosti sortirana. Ono što se događa pri korištenju *bubble sorta* je da pri svakom prolazu na kraj liste dolaze najveći elementi. Tako će u prvom prolazu, najveći element liste doći na zadnje mjesto, u drugom će prolazu drugi najveći doći na predzadnje mjesto i tako redom. Ti elementi zapravo „izranjaju“ na kraj liste poput mjehurića zbog čega se ovaj način sortiranja zove *bubble sort*. Iako je vrlo jednostavan te se lako može sastaviti, nema veliku korist u stvarnome životu zato što nije efikasana sam proces može biti vrlo spor i dugačak, posebice kod većih nizova. Ipak, svoju svrhu nalazi uglavnom u manjim nizovima te kao metoda sortiranja kod gotovo potpuno sortiranih nizova u kojima ima samo još nekoliko elemenata na krivim mjestima.

Iako zbog svoje jednostavnosti ima svrhu, neki ljudi misle da i tu nije koristan kao što piše u u sljedećem paragrafu „Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students. However, some researchers such as Owen Astrachan have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.“[2](Wikipedia, 1.9.2021.):

Primjer *bubble sorta*:

Prvi prolaz

Prvo algoritam uspoređuje prva dva broja. Kako je 5 veći od 1, tako se mjenjaju:

(**5** **1** 4 2 8) –> (1 5 4 2 8)

(1 5 4 2 8) –> (1 4 5 2 8) 5 i 4 se uspoređuju i mijenjaju jer je 5 veći i algoritam skače na sljedeće brojeve.

(1 4 5 2 8) –> (1 4 2 5 8) 5 je veće od 2 pa se mjenjaju i ide dalje.

(1 4 2 5 8) –> (1 4 2 5 8) 5 i 8 su pravilno poredani pa se ništa ne mijenja, ali niz i dalje nije sortiran pa ide u drugi prolaz

Drugi prolaz:
(1 4 2 5 8) –> (1 4 2 5 8) 1 i 4 ostanu na svojim mjestima jer su dobro sortirani.
(1 4 2 5 8) –> (1 2 4 5 8), 4 i 2 se mijenjaju jer je 4 veći.
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8) 4 i 5 su na svojim mjestima i ništa se ne mijenja.

[2] „Zbog svoje jednostavnosti, bubble sort često se koristi za uvođenje koncepta algoritma ili algoritma za sortiranje učenicima informatike. Međutim, neki istraživači, poput Owena Astrachana, uložili su velike napore kako bi umanjili značaj bubble sorta i njegovu važnost i popularnost u području informatičkog obrazovanja, preporučujući da se više niti ne podučava“. (Prijevod autora)

(1 2 4 5 8) –> (1 2 4 5 8) ista stvar sa 5 i 8.

Niz je sortiran,ali *bubble sort* funkcionira tako da i kad se sortira mora još jednom proći kroz niz bez ikakvog mijenjanja zbog provjere da je sve dobro i onda algoritam staje. Treći prolaz

(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Algoritam je gotov i niz je sortiran.

2.12. Implementacija BUBBLE SORTa u C++

Primjer implementacije *bubble sorta* u C++-u:

```
// Funkcija za implementaciju bubble sorta
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)
   // Zadnji element je već na svom mjestu
   for (j = 0; j < n-i-1; j++)
      if (arr[j] > arr[j+1])
         swap(&arr[j], &arr[j+1]);
}
```

3. USPOREDBA ALGORITAMA

U ovom je radu opisan rad algoritama, način na koji funkcioniraju, njihova brzina i efikasnost. Do sada nisu detaljnije bili objašnjeni razlozi zbog kojih su neki brži, a neki sporiji, koji se kriteriji gledaju pri rasuđivanju brzine i efikasnosti algoritama te kako se algoritmi uspoređuju. Već je prije spomenuto da je *Bubble sort* najsporiji algoritam koji nema puno upotrebne koristi, dok je *Quick sort* najbrži pa je tako i efikasniji.

U ovom će dijelu rada biti objašnjeno zašto je tomu tako. Pri uspoređivanju vremena odvijanja algoritma, uspoređuju se tri slučaja – prosječan slučaj, najgori slučaj i najbolji slučaj. Najbolji je način za usporedbu algoritama efikasnost algoritama. To se uspoređuje s brojem elemenata u nizu koji se moraju sortirati te s povećanjem broja elemenata. Što je više elemenata u nizu, to je teže sortirati niz te je cijeli proces dugotrajniji. Neki algoritmi poput *Bubble sorta*ne funkcioniraju dobro pri sortiranju dugačkih nizova, dok su neki poput *Shell sorta* odlični u tome. Kako bi se izrazila efikasnost algoritama koristi se oznaka O. Ona ovisi o tome kako se ponaša vrijeme izvođenja algoritama (raste li ili pada) te broj elemenata u tom prostoru. O(n) je oznaka koja izražava vrijeme potrebno da se izvrši zadatak (trajanje obavljanja rada algoritma), odnosno vremensku kompleksnost algoritma. Vremenska se kompleksnost računa tako da se gleda broj operacija koje algoritam izvršava, primjerice, zamjene elemenata ili prolazi reda. Svaka ta operacija ima određeno trajanje. Prema tome, uzima se da se količina vremena i broj elementarnih operacija koje izvodi algoritam razlikuju za najviše konstantan faktor. Najgori je slučaj situacija u kojem je algoritmu potrebno najduže vrijeme da se izvrši.

Scenarij prosječne kompleksnosti rjeđi je slučaj zato što je to specifična situacija i postoji samo određeni broj mogućnosti te je teško izračunati prosjek zbog raznih faktora niza - povećanja niza dodavanjem elemenata i slično. Najbolji je slučaj onaj slučaj u kojem je algoritmu potrebno najkraće moguće vrijeme kako bi se izvršio. Oznaka kompleksnosti vremena može biti O(n) ili O(2^n) ili razne oznake, gdje je „n“ input prikazan u elementima od kojih se niz sastoji.

Ponekad neki algoritmi imaju jednaku vrijednost za efikasnost, no njihove su brzine upisivanja podataka, odnosno input, tada najčešće drugačije. Neki algoritmi odlično funkcioniraju s određenim inputima, a loše s drugima. Dobar je primjer *Quick sort* koji ovisi o inputu podataka, dok primjerice kod *Merge sorta* unos podataka uopće ne utječe na njegovu efikasnost.

Kod usporedbe algoritama, također je vrlo važna njihova stabilnost.

3.1. O klasa

Objašnjenje tipova O klasa:

O(1)– Konstantno vrijeme

Ovo znači da je izvršavanje algoritma uvijek konstantno i neovisno o inputu, odnosno povećanju broja elemenata u nizu.

O(n)– Linearno vrijeme

Ova kompleksnost znači da se vrijeme linearno mijenja u odnosu na povećanje elemenata u nizu. Dakle, ako se niz poveća za duplo elemenata, vrijeme izvršavanja se povećava za duplo.

O(n²) – Kvadratno vrijeme

Vrijeme izvršavanja se mijenja linearno s kvadratom broja elemenata u nizu.
Dakle ako je input 4, vrijeme će se povećati za 16.

Primjeri: *insertion sort, bubble sort*

O(log n) – Logaritamsko vrijeme

Ovo vrijeme funkcionira tako da se vrijeme mijenja za konstantnu veličinu kada se broj elemenata u inputu poveća dvostruko. Na primjer, ako se vrijeme promijeni za 30 sekundi kada se sa 1000 elemenata u nizu prijeđe na 2000 elemenata u nizu, sa 2000 na 4000 će se opet promijeniti iako je razlika puno veća nego u prvom slučaju, ali isto tako će se za 30 sekundi promijeniti kada prijeđe s 4000 na 8000. Primjer algoritma koji koristi ovu vremensku kompleksnost je Algoritam binarnog pretraživanja.

O(n * log n) – Kvazilinearno vrijeme

Kvazilinearno vrijeme je slično logaritmičkom samo što promjena nije konstantna, nego se linearni dio(n) množi sa logaritamskim dijelom(log n). Dakle, u početku se vrijeme dosta mijenja, ali što duže ide je konstantnije i manje se mijenja.

Kvazilinearno vrijeme se obično pojavljuje kod efikasnijih algoritama kao *quick sort, merge sort* i *heap sort*.

Prema kompleksnosti bi se te klase ovako podijelile, od „najjednostavnije“ do najkompleksnije:

- O(1) – Konstantno vrijeme
- O(log n) – Logaritamsko vrijeme
- O(n) – Linearno vrijeme
- O(n log n) – kvazilinearno vrijeme
- O(n²) – Kvadratično vrijeme

3.2. Prikaz slučajeva

Tablica 1: Prikaz slučajeva

Algoritam	**Najbolji slučaj**	**Prosječni slučaj**	**Najgori slučaj**
BUBBLE SORT	O(n)	O(n^2)	O(n^2)
SHELL SORT	O(n^2)	O(n*log n)	O(n*log n)
MERGE SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
INSERTION SORT	$O(n)$	$O(n\ 2\ )$	$O(n\ 2\ )$
HEAP SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QUICK SORT	$O(n \log n)$	$O(n\ 2\ )$	$O(n \log n)$

Izvor: Izrada autora

Dakle, vremenska kompleksnost objašnjava kako i koliko se brzina izvršavanja algoritma mijenja ovisno o broju elemenata u inputu i kako se kroz to mogu uspoređivati elementi.

Usporedba algoritama:

Najjednostavniji algoritam je *Bubble sort* i to je u isto vrijeme algoritam koji se najmanje koristi. Iako može biti jako koristan u situacijama gdje niz ima malo elemenata ili je gotovo sortiran.

Insertion sort je malo bolji kada gledamo slučajeve. U najboljem slučaju je niz već sortiran i algoritam samo provjerava niz s lijeva na desno, element po element tako da je vremenska kompleksnost O(n). A u najgorem slučaju se svaki element mora zamjenjivati, odnosno vremenska kompleksnost mu je O(n^2) što je još uvijek dobro.

Merge sort je algoritam koji funkcionira na „divide and conquer“ principu. Kako ovaj algoritam funkcionira tako da se niz podijeli na dva niza, koji se svaki sortira i onda se vežu natrag u sortirani niz, onda su i najgori i najbolji slučaj isti, odnosno O(n log n)

Quick sort je jedan od najbržih i najkorisnijih algoritama. Smatra ga se najboljim algoritmom za sortiranje podataka, najviše zbog svog prosječnog vremena koje je O(n log n), dakle prosječno vrijeme od *quick sorta* ujedno je i njegovo najbolje vrijeme.

Heap sort je jako efikasan algoritam kojemu se vrijeme mijenja logaritamski tako da se vrijeme izvršavanja ne usporava s novim elementima. Jednostavan je za koristiti u usporedbi s algoritmima slične efikasnosti, a najgore, prosječno i najbolje vrijeme su mu svi dobri, tako da je dobar algoritam za sustave gdje je važna konzistentnost i nema mjesta za pogreške.

Shell sort je vrlo efikasan algoritam koji je osnovan na temelju *insertion sorta*. Samo što je poboljšan i efikasniji. Međutim, također je kompleksan za korištenje. Koristan je u određenim situacijama kao, primjerice kod niza koji je gotovo sortiran ili kada se prvi i zadnji element trebaju promijeniti. I u takvim situacijama je brži od većine drugih algoritama.

4. ZAKLJUČAK

U vrijeme korištenja prvih algoritama, bili su korišteni samo za matematiku, ali s vremenom su se pronašli načini za razne uporabe, a i algoritmi su se razgranali ovisno o tome za što se koriste, tako su nastali algoritmi za sortiranje podataka.

U današnje vrijeme kada se podaci šire kao nikad prije, važno nam je znati kako ih najefikasnije sortirati. Posebno to vrijedi za tvrtke, bolnice, knjižnice i slične organizacije gdje se koriste velike količine podataka. Zbog toga je važno znati razne algoritme koji se mogu koristiti kako bi se izradili programi koji izvode za organizaciju podataka. I kao što je vidljvio u ovom radu, razni algoritmi su korisni u određenim uvjetima. Stoga ne možemo tvrditi da je neki algoritam najbolji ili najbrži, međutim uz dovoljno znanja o svakom možemo ustvrditi koji je najbolji za pojedinu specifičnu situaciju. Iako neki jesu „lošiji“ od drugih i sporiji, kao *bubble sort*, oni i dalje imaju svoju korist, kao u ovom slučaju za edukaciju o algoritmima. Prema autoru jednog teksta s Interneta: „Svaki od ovih algoritama ima neke prednosti i nedostatke i može se učinkovito odabrati ovisno o količini podataka kojima se treba rukovati.“ (A.M., 2020, PC Cheap)

5. POJMOVNIK

Pojmovi koje se ne rabe često a koriste se u ovom radu su sljedeći:

Niz – skup brojeva

Input – prazno mjesto u nizu u koje se stavlja neki element

Output – Mjesto iz kojeg se vadi neki element

C++ - Programski jezik koji će se u ovom radu koristiti za primjere implementacije algoritama

Algoritam – skup definiranih naredbi

6. IZJAVA

Izjava o autorstvu završnog rada i akademskoj čestitosti

Ime i prezime studenta: Vlaho Vručinić

Matični broj studenta: 6-124/17

Naslov rada: Pregled i analiza algoritama za sortiranje konačnog niza realnih brojeva

Pod punom odgovornošću potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada.

Potvrđujem da je elektronička verzija rada identična onoj tiskanoj te da je to verzija rada koju je odobrio mentor.

Datum	Potpis studenta
_____	_____

7. POPIS LITERATURE

7.1. Članci i završni radovi

Ivan Reif. (2011.) *Vizualizacija algoritama sortiranja*. Završni rad fakulteta organizacije i informatike Varaždin, Sveučilišta u Zagrebu.
https://bib.irb.hr/datoteka/526116.Vizualizacija_algoritma_sortiranja.pdf

Andrijana Ivančić. (2018.) *Usporedba algoritama za sortiranje i pretraživanje*.Završni rad Sveučilišta Jurja Dobrile u Puli.
file:///C:/Users/Vlaho/Downloads/ivancic_andrijana_unipu_2018_zavrs_sveuc.pdf (ovo je u pdfu i nece se moci otvoriti na drugim kompjuterima, pogledati sto mogu)

Krešimir Šuljug. (2017.) *Paralelne izvedbe algoritama za sortiranje*. Završni rad Sveučilišta Joispa Jurja Strossmayera u Osijeku.
https://repozitorij.etfos.hr/islandora/object/etfos%3A1180/datastream/PDF/view

7.2. Internetski izvori

Wikipedija. (n.d.) *Algoritam*. Preuzeto s
https://hr.wikipedia.org/wiki/Algoritam (3. lipnja 2021.)

GeeksforGeeks. (n.d.). *Sorting Algorithms*. Preuzeto s https://www.geeksforgeeks.org/sorting-algorithms/ (31. srpnja 2021.)

Tutorialspoint. (n.d.) *Data Structure and Algorithms Tutorial*. Preuzeto s
https://www.tutorialspoint.com/data_structures_algorithms/index.htm (25. lipnja 2021.)

AfterAcademy. (25. prosinca 2019.) *Comparison of Sorting Algorithms*. Preuzeto s
https://afteracademy.com/blog/comparison-of-sorting-algorithms (23. srpnja 2021.)

Cprogramming.com. (n.d.) *Sorting Algorithm Comparison*. Preuzeto s
https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html (15. srpnja 2021.)

TiTrias. (n.d.) *Usporedba brzina algoritama sortiranja*. Preuzeto s
https://loomen.carnet.hr/pluginfile.php/3432612/mod_resource/content/2/3_26_Usporedba%20brzina%20algoritama%20sortiranja.pdf (17. srpnja 2021.)

HappyCoders.eu (28. svibnja 2021.) *Big O Notation and Time Complexity*. Preuzeto s
https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/#O1_-_Constant_Time (30. srpnja 2021.)

Digit.in(28. kolovoz 2021.)*The Origin and Evolution of Algorithms*. Preuzeto s
https://www.digit.in/features/science-and-technology/the-origin-of-algorithms-30045.html

pcchip.hr(26. kolovoza 2021.)*5 algoritama sortiranja koje svaki programer treba znati*. Preuzeto s: https://pcchip.hr/softver/korisni/5-algoritama-sortiranja-koje-svaki-programer-treba-znati/

8. POPIS SLIKA, GRAFIKONA I TABLICA

Popis tablica:

ŽIVOTOPIS

PERSONAL INFORMATION

Vlaho Vručinić

Božidara Adžije 22, HR-10 000 Zagreb, Hrvatska

++ 385 1 3646 179 ++ 385 99 6711 543

vrucinicvlaho@gmail.com

Sex Muško | Date of birth 26/02/1999 | Nationality Enter nationality/-Hrvat

JOB APPLIED FOR / POSITION / PREFERRED JOB / STUDIES APPLIED FOR

Poslovi povezani s bazama podataka – izrada i/ili održavanje

WORK EXPERIENCE

2018- danas

Asistent u financijama – studentskiposao

Centar za civilneinicijative, Zagreb, www.cci.hr

- Knjiženje ulaznih računa
- Praćenje potrošnje sredstava po donatorima
- Izrada baza podataka za pojedine projekte poslodavca

EDUCATION AND TRAINING

2017 – 2021

Informacijske tehnologije - apsolvent

Replace with European Qualification Framework (or other) level if relevant

Veleučilište Baltazar, Zaprešić

PERSONAL SKILLS

Mother tongue(s): Hrvatski

Other language(s)

	UNDERSTANDING		SPEAKING		WRITING
	Listening	Reading	Spoken interaction	Spoken production	
Engleski	C1	C 1	C 1	B 2	B 2
	Replace with name of language certificate. Enter level if known.				
Fransucki	C 1	B 2	B 2	**B 2**	B 1
	Replace with name of language certificate. Enter level if known.				

Levels: A1/2: Basic user - B1/2: Independent user - C1/2 Proficient user
Common European Framework of Reference for Languages

Communication skills	▪ Dobre komunikacijske vještine stečene volontiranjem i studentskim poslovima u udrugama
Organisational / managerial skills	Organiziran sam u poslu i zadacima kojedobivam i preuzimam kao svoju odgovornost ▪ Preferiram individualan rad i odgovornost
Job-related skills	▪ Iskustvo u izradi baza podataka organizacije glazbenih događaja ▪ Iskustvo financijskog praćenja EU projekata u neprofitnom sektoru ▪ Iskustvo knjiženja ulaznih računa udruge ▪ Iskustvo organizacije glazbenih događaja
Computer skills	▪ Dobro poznavanje Microsoft Office™ paketa, odlično korištenje Excel programa
Other skills	Poučavanje starijih osoba upotrebi digitalnih alata Izrada skripti za osobe treće dobi u području primjene društvenih mreža Održavanje web stranica
Driving licence	Ne